VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

DIGITÁLNÍ STEGANOGRAFIE PRO SPUSTITELNÉ SOUBORY

DIGITAL STEGANOGRAPHY FOR EXECUTABLES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PAVEL TOBIÁŠ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JOSEF STRNADEL, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Tobiáš Pavel**
Obor: Informační technologie
Téma: **Digitální steganografie pro spustitelné soubory**
Digital Steganography for Executables
Kategorie: Bezpečnost

Pokyny:

1. Vytvořte přehled metod z oblasti digitální steganografie pro skrývání informace ve spustitelných souborech (dále jen "steganografie"), jejich vlastností a shrňte současný stav v této oblasti.
2. Zvolte typ skrývané informace (textová, obrazová apod.) a její vlastnosti. Na základě existující či vlastní analýzy formátů spustitelných souborů a typu skrývané informace zužte a zvolte vhodné formáty spustitelných souborů a vhodné metody pro steganografii.
3. Implementujte několik existujících metod steganografie, zvažte jejich modifikace, popř. návrh a implementace vlastních metod.
4. Demonstrujte a vyhodnoťte funkčnost a vlastnosti implementovaných metod z hlediska skrývání a odkrývání zvoleného typu informace.
5. Dosažené výsledky diskutujte, navrhněte možné návaznosti a rozšíření předloženého řešení.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Strnadel Josef, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá steganografickým ukrýváním libovolných dat do spustitelných souborů. Nejprve hovoří obecně, zejména o injekčních a substitučních steganografických metodách nad různými typy krycích objektů. Poté se zaměřuje na spustitelné soubory formátu ELF a rodinu instrukčních sad x86; zmiňuje permutační metody nad instrukcemi a řetězy základních bloků a dopodrobna rozebírá substituční metodu ekvivalenčních tříd instrukcí. Následně je popsán návrh, implementace, testování a optimalizace vlastního řešení založeného na poslední ze zmíněných metod. Posléze jsou popsány metody a výstupy experimentů s vlastním řešením.

Abstract

This bachelor's thesis concerns itself with steganographic concealment of arbitrary data in executable files. Initially it speaks in general terms, mainly about injection- and substitution-based steganographic methods for various types of cover-objects. Afterwards, the focus is on executable files in the ELF format and the x86 ISA family; permutation-based methods for instructions and basic block chains are mentioned and the substitution-based method of instruction equivalence classes is examined. Consequently, the design, implementation, testing and optimization of a custom solution based on the last mentioned method are described. Finally, the methods and outcomes of experimenting with the custom solution are described.

Klíčová slova

steganografie, spustitelný soubor, ELF, x86, instrukce, substituce, ekvivalenční třída

Keywords

steganography, executable, ELF, x86, instruction, substitution, equivalence class

Citace

TOBIÁŠ, Pavel. *Digitální steganografie pro spustitelné soubory.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

Digitální steganografie pro spustitelné soubory

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Tobiáš
17. května 2018

Poděkování

Děkuji vedoucímu bakalářské práce panu Ing. Josefu Strnadelovi, Ph.D. za užitečnou metodickou a pedagogickou pomoc při zpracování práce.

Obsah

Kapitola 1

Úvod

Tato práce pojednává o steganografii: vědě a umění skrývání tajných dat v jiných – nosných – datech za účelem utajené komunikace. Konkrétně se práce zabývá touto problematikou v doméně spustitelných souborů jakožto oněch nosných dat. Kromě rozboru teoretických principů práce plní i funkci dokumentace ukázkové aplikace, jež steganografické ukrývání dat ve spustitelných souborech implementuje.

Úvodní kapitola práce čtenáře seznámí se stěžejními koncepty disciplíny, základní terminologií a jejím historickým vývojem. Dále zde steganografie bude zasazena do širšího oborového kontextu a srovnána se souvisejícími disciplínami. V neposlední řadě se kapitola pokusí o smysluplnou klasifikaci steganografických metod, uvede jejich konkrétní příklady a detailně vysvětlí ty z obecných principů, jež je třeba chápat pro porozumění zbývající části textu.

Následující, třetí kapitola se věnuje steganografii již v kontextu spustitelných souborů. Nejprve obeznámí čtenáře s formátem spustitelných souborů, na nějž se práce zaměřuje. Poté pohovoří o množství steganografických metod, některé z nich do detailu popíše a zmíní jejich existující implementace.

Kapitoly 4 a 5 dokumentují celý proces vývoje výše zmíněné demonstrační aplikace, tedy její návrh, implementaci, testování, optimalizaci a hodnocení z hlediska smysluplných steganografických kritérií.

Kapitola 2

Steganografie obecně

Tato kapitola seznámí čtenáře s oborem steganografie po obecné stránce; definuje pojem steganografie, pohovoří o historickém vývoji disciplíny, uvede příklady její praktické aplikace v současnosti a zmíní se o spojitosti disciplíny s jinými obory. Dále bude čtenář obeznámen ze základní terminologií steganografie potřebnou pro pochopení následujícího textu práce. V neposlední řadě kapitola hovoří o klasifikaci steganografických metod dle dvou různých kritérií a uvádí jejich výhody a úskalí na obecné rovině i na konkrétních příkladech.

2.1 Definice pojmu

Obecně lze steganografii (z řec. *steganos* – skrytý, *graphein* – psát [3]) popsat jako studium technik, jejichž cílem je co nejvýhodnějším způsobem využívat veřejných komunikačních kanálu jako médií pro skrytý přenos privátních zpráv [16].

Jako model ilustrující pointu steganografie se často používá tzv. *problém vězňů* [17]: dva vězni zavření v oddělených celách se snaží naplánovat společný útěk z vězení. Avšak jediný způsob, jak mohou komunikovat, je výměna zpráv přes jejich dozorce. Ten samozřejmě vidí obsah zpráv a nedovolí žádnou podezřelou komunikaci včetně šifrované. Vězni tedy musí najít způsob, jak ukrýt svoje dorozumívání do zdánlivě bezúhonného obsahu, například nevinně vypadajících „malůvek“, do jejichž rozložení tvarů či barev jsou zakódovány tajné zprávy. Obecně řečeno tedy steganografie nachází uplatnění hlavně v situacích, kdy hrozí nějaký postih za objevenou komunikaci.

V její současné, digitální podobě lze steganografii považovat za disciplínu teoretické informatiky [13], jehož předmětem je zkoumání způsobů, jak do souborů, popřípadě datových proudů, různých formátů ukrývat jiná data co největšího objemu s co nejmenším rizikem jejich objevení. Toho lze obecně docílit nalezením nějakého druhu redundance běžně se vyskytující v daném formátu a jejího využití k zakódování skrývaných dat, aniž by byla porušena sémantika nosných dat v kontextu jejich běžné interpretace [4].

2.2 Historie disciplíny

Steganografie má překvapivě dlouhou historii – mnohem delší než její „adoptivní“ mateřský obor, teoretická informatika. První zdokumentované techniky ukrývání tajných zpráv do různých fyzických objektů lze hledat už ve starověku.

2.2.1 Starověk

Antický historik Hérodotos z Halikarnássu (5. století př. n. l.) ve svém spisu *Histories apodeixis* popisuje využití voskových destiček jako de-facto steganografického média. Standardně se tyto hladké kusy dřeva používaly pro uchovávání viditelného písemného projevu vyrytého do vrstvy vosku, jíž byly pokryty. Avšak Demaratus, bývalý spartský panovník vyhnaný do Perské říše, který se rozhodl navzdory vyhnanství varovat svoji rodnou zemi o tajně plánované vojenské invazi Peršanů, vyryl kýženou zprávu přímo do dřevěného podkladu před jeho zalitím čerstvou vrstvou vosku. Destičky tak vypadaly jako běžné zboží, které se bez problému dostalo do Sparty, a tajná zpráva byla objevena manželkou současného panovníka Leonida [13].

Další příklady podobných technik užívaných v antickém Řecku, především pro účely utajené militární komunikace, zdokumentoval historik Aeneas Tacticus (4. století př. n. l.) ve svém díle o umění války. Sám navrhl tzv. *astragal* – dutou kouli s nepopsanými otvory tajně reprezentujícími písmena, jejichž pořadí čtení indikoval provázek jimi postupně propletený [8].

Pro další steganografický milník hodný zmínky není třeba opouštět antiku, v 1. stol. nl. byl totiž filosofem Pliniem Starším objeven neviditelný inkoust (mléčná esence extrahovaná z rostliny *tithymalus*) nečitelný po zaschnutí a permanentně znovuzčitelnělý zahřátím [13].

2.2.2 Novověk

První známé užití termínu steganografie se datuje až do roku 1499. Učinil ho Benediktýnský mnich Johannes Trithemius ve svém nejvýznamnějším díle *Steganographia* kuriózním tím, že o steganografii nejen informuje, ale je navíc samo její aplikací. Běžně čitelný obsah této trojsvazkové knihy se totiž zabývá magií a okultními praktikami; pojednání o steganografii a kryptografii je do něj samo o sobě steganograficky zakódováno [6].

Za zmínku stojí několik dalších novověkých posunů disciplíny. v roce 1550 navrhl italský polyhistor Girolammo Cardano tzv. *Cardanovu mřížku* – destičku z pevného materiálu proděrovanou na příslušných místech tak, aby po přiložení k papíru pokrytému nevinně vypadajícím textem odkryla pouze znaky nesoucí tajnou zprávu [17]. Italský vědec Giovanni Porta (narozen 1535) kontribuoval ke steganografii návrhy různých nových technik, například vynálezem neviditelného inkoustu na lidskou kůži a metodoy psaní na bílek uvařeného vejce skrze skořápku pomocí tinktury z ledku a octa [13].

2.2.3 Moderní dějiny

Co se týče předinformační etapy moderních dějin, steganografie byla hojně využívána především pro utajenou militární komunikaci v prvních dvou světových válkách. Běžné bylo používání tzv. *žargonové šifry* [13] spočívající v substituci slov za slova jiného významu z nesouvisejících kontextů (křižník → vazelína, torpédoborec → parafín a podobně ⇒ válečná korespondence vypadala jako zpráva o drogistickém sortimentu).

Dále se využívalo tzv. *otevřených kódů* (*null ciphers*) [17] založených na čtení konkrétních znaků krycího textu předem domluveným způsobem (například přečtením prvního písmene každého slova v textu „fandím i těm v uniformách trestanců“ dostáváme skrytou zprávu „fitvut“).

V neposlední řadě stojí za zmínku metoda *mikroteček* [17] založená na tisku textového či grafického obsahu zmenšeného na velikost tečky vyprodukované běžným psacím strojem,

jeho užití jako interpunkčního znaménka v neškodně vyhlížející korespondenci a následném čtení pomocí mikroskopu.

2.2.4 Informační věk

Od počátku informačního věku má smysl se steganografií zabývat v zásadě výlučně v doméně digitálních dat [13]. Existují důkazy o aktivním užívání různých datových formátů jako nosičů tajné komunikace po internetu, například bezpečnostními složkami států a teroristickými organizacemi, ale také civilisty sdílejícími různý trestně postihnutelný obsah. Oblíbené jsou především různé formáty obrazových dat; zaprvé kvůli relativně velké kapacitě (viz dále), zadruhé kvůli jejich nenápadnosti na sociálních sítích, image-boardech a podobných webových službách [5].

2.3 Základní terminologie

Krycí objekt (*cover object*) [24] je soubor nebo datový proud sloužící jako médium pro ukrytí tajné zprávy.

Stegogram (*stego-object*) [13] je krycí objekt již obsahující ukrytou zprávu.

Stegosystém (*stegosystem*) [13] je konkrétní implementace nějaké steganografické metody (či kombinace více metod) pro určitou množinu krycích objektů. Zahrnuje *vkládací* (*embedding ~*) a *exktrakční algoritmus* (*extraction algorithm*) [24], tedy algoritmy sloužící k zakódování a extrakci tajné zprávy.

Steganografický klíč (*stego-key*) [24] je volitelný dodatečný vstup do výše zmíněných algoritmů ovlivňující způsob vložení zprávy a potřebný k její úspěšné extrakci.

2.4 Relevantní vlastnosti a metriky

Kapacita krycího objektu [15] je maximální bitová délka ukrývaných dat, kterou lze do daného krycího objektu vložit. Je potřeba ji uvažovat vždy v kontextu konkrétního stegosystému, protože různé metody a jejich implementace mohou různě efektivně využívat potenciální kapacity krycích objektů.

Robustnost stegosystému [15] vyjadřuje odolnost skrytých dat vůči poškození určitou transformací stegogramu. v případě obrazových dat jako krycího objektu lze uvažovat například ztrátovou kompresi a transformaci domény z prostorové do frekvenční či naopak.

Imperceptibilita ukrytých dat [24] je vlastnost stegogramu nevzbudit nahodilé podezření u uživatele krycího objektu, kterému není určena tajná zpráva v něm ukrytá [24]. Imperceptibilitu ukrytých dat mohou ohrozit různé druhy netradičního chování krycího objektu vzniklé nedostatečně uváženými metodami ukrývání nebo jejich parametry, například:

- různé vnímatelné artefakty v audiovizuálních krycích datech (šum, ozvěna, ...);
- podezřele dlouho trvající síťový přenos krycích paketů;
- podezřelé výkonnostní změny programu načteného z krycího spustitelného souboru.

Nedetekovatelnost stegogramu [10] je jeho odolnost proti různým stegoanalytickým [1] metodám aktivně usilujícím o objevení ukryté zprávy.

2.5 Souvislost s jinými obory

Disciplína, která má se steganografií asi nejzřetelnější souvislost, je **kryptografie**. Zásadní rozdíl mezi nimi tkví v tom, že zatímco kryptografické techniky slouží k utajení obsahu tajné zprávy, respektive znemožnění jeho čtení bez vlastnictví dešifrovacího klíče, ty steganografické se snaží o utajení její samotné existence. Navíc, na rozdíl od kryptografických systémů, u stegosystémů není nutně žádoucí, aby se samy o sobě řídily *Kerckhoffsovým principem*[2] – mohou být implementací tzv. *bezpečnosti skrze utajení (security through obscurity)* – z kteréhož důvodu se steganografie s kryptografií někdy kombinuje: zpráva se nejprve zašifruje pro případ, že bude objevena, a poté ukryje do krycího objektu [9].

Souvislost lze rovněž pozorovat mezi steganografíí a **digitálním vodotiskem**. v obou případech se jedná o kódování dodatečných informací do existujících dat a některé techniky mohou být vhodné pro obojí. Podstatnými odlišnostmi jsou:

- **motivace k použití:** vodotisk slouží k podepsání objektu z důvodu identifikovatelnosti jeho původu, nikoliv k tajné komunikaci;
- **vztah nosných dat s ukrytými:** vodotisk nese informace přímo související s jeho nosičem, zatímco význam steganograficky kódované zprávy je typicky absolutně nesouvisející s krycím objektem;
- **míra relevance jednotlivých vlastností:** z výše popsaných metrik hraje ve steganografii největší roli kapacita a imperceptibilita; u vodotisku je nejdůležitější robustnost – ideální je, když při pokusu o odstranění vodotisku dojde ke znehodnocení jím podepsaných dat.

2.6 Klasifikace metod

Nejtriviálnějším kritériem dělení steganografických metod je typ krycích objektů, jehož využívají. Lze pak hovořit například o [24]:

- **textové steganografii** využívající literárních děl;
- **obrazové steganografii** využívající statických či dynamických obrazových dat;
- **síťové steganografii** využívající síťových rámců, paketů, datagramů či zpráv;
- **steganofonii** [8] využívající digitálního záznamu zvuku;
- **steganografii spustitelných souborů**, jíž se dopodrobna zabývají další kapitoly této práce

...a tak dále. O něco zajímavějším klasifikačním kritériem je základní princip kódování dat do krycích objektů. Můžeme pak hovořit o *injekčních*, *substitučních* a *generačních* metodách.

[1] Stegoanalýza (*steganalysis*) je obor zabývající se metodami detekce přítomnosti steganograficky ukrytých informací [13].

[2] „Bezpečnost šifrovacího systému má záviset pouze na utajení klíče, tedy prozrazení principu šifrování nemá bezpečnost systému ohrozit [20].“

2.6.1 Injekční metody

Injekční metody (*injection-based methods*) [8] jsou založeny na připojení ukrývaných dat k datům krycího objektu na místo, kde neovlivní jeho běžnou interpretaci. Původní data tedy zůstanou nedotčena, ale velikost výsledného stegogramu je oproti původnímu krycímu objektu větší minimálně o velikost ukrývaných dat.

Hlavní výhodou injekčních metod je, že původní data nejsou jakkoliv modifikována. To je výhodné například:

- v momentě, kdy neexistuje způsob, jak změnit jejich zápis tak, aby jejich sémantika zůstala stejná (čili nelze využít dále popsaných substitučních metod);
- probíhá-li kontrola integrity dílčích segmentů krycích dat (například kontrolní součet) a jiná, byť sémanticky shodná, reprezentace dat by kontrolou neprošla.

Další, spíše teoretickou, výhodou těchto metod je, že z jejich hlediska krycí objekty mohou mít teoreticky neomezenou kapacitu, je však třeba mít na paměti, že zvětšováním objemu stegogramu oproti původnímu krycímu objektu roste jeho podezřelost.

Příklady injekčních metod

Serializační formáty jako *XML*, *JSON* a další (běžně používané například pro komunikaci mezi klientskými a serverovými částmi webových aplikací) lze použít díky jejich relativní strukturální flexibilitě. Serializovaná stromová struktura je typicky příjemcem označena za validní, obsahuje-li všechny povinné očekávané uzly; ty navíc jsou parserem jednoduše ignorovány, ergo jejich použití k injekčnímu ukrytí dat neovlivní běžnou interpretaci krycího objektu.

Formátovaný text – instance formátů využívaných textovými procesory (*RTF*, *OpenDocument*, ...), aplikace značkovacích jazyků (*HTML*, *Markdown*, ...) apod. mohou obsahovat nevyužité formátovací příkazy (například prázdné formátovací značky v *HTML*), nevykreslovaná metadata (například `<meta>` *značky* a komentáře v *HTML*), text skrytý vhodným formátováním a další data navíc, imperceptibilní při běžném užití, s potenciálem kódovat ukrývaná data.

Audiovizuální data – soubory se vzorkovanými obrazovými daty či audiem zpravidla začínají hlavičkou, která obsahuje informaci o délce užitečných dat v bajtech, pixelech, vzorcích apod. Jakákoliv data nacházející se za nimi, třeba ta, jež chceme do nich ukrýt, jsou interprety těchto formátů typicky ignorována.

Moderní multimediální kontejnery jsou dalším vhodným kandidátem. Například kontejnery formátu *Matroska* [21] dokážou uchovávat kromě stop s videem, audiem, titulky, ap. také obecné *přílohy* (*attachments*) libovolného typu. Jejich zamýšleným užitím jsou například obrázky s CD obaly či fonty potřebné ke správnému zobrazení titulků, nicméně není problém přiložit jakákoliv jiná, potenciálně nesouvisející, data.

2.6.2 Substituční metody

Substituční metody (*substitution-based methods*) [8] jsou založeny na systematickém nahrazování částí krycích dat za jiná data tak, aby byly zachovány všechny důležité vlastnosti krycího objektu v kontextu jeho běžné interpretace. Mohou, mimo jiné, využívat:

- nevyužitých míst v souborových strukturách, jako jsou:
  - zarovnávací bajty;
  - položky vyhrazené pro užití v budoucích verzích formátu;
  - standardem definované, ale v praxi ignorovaná, položky

  ...a podobně (příkladem jsou třeba položky `Reserved1`, `Reserved2` a `BitmapOffset` v hlavičce *BMP* souborů [2]);
- nedokonalosti lidského zraku a sluchu, které činí nuance v obrazové a zvukové reprodukci imperceptibilními (příkladem je dále zmíněná *LSB-metoda*);
- vlastností daného krycího objektu dovolujících reprezentovat jednu informaci více různými posloupnostmi bitů.

Ekvivalenční třídy

Především v souvislosti s posledním zmíněným bodem je třeba zmínit následující skutečnost: Přestože může být několik různých bitových sekvencí na daném místě ekvivalentních a jejich libovolná záměna nealterovat význam krycího objektu, nemusí se nutně jednat o všech 2^n kombinací n bitů. v těchto případech je třeba mít na paměti, že:

1. maximální bitová délka dat, která do sekvence bitů ukrýváme, neodpovídá bitové délce sekvence, nýbrž její reálné entropii;
2. ukrývaná data nemohou být přímo bitově reprezentována sama sebou, nýbrž kódovaná nějakým alternativním způsobem.

Jedno z možných řešení problému je formulace ekvivalenčních tříd daných bitových posloupností. Jedna taková *ekvivalenční třída* je množina všech posloupností bitů, které jsou v daném kontextu libovolně zaměnitelné. Každý prvek této množiny pak může reprezentovat libovolnou posloupnost bitů. Členy ekvivalenční třídy o mohutnosti m lze například libovolně seřadit, indexovat od 0 do $m - 1$ a deklarovat, že každý z nich kóduje binární reprezentaci svého indexu na $\log_2 m$ bitech.

Toto řešení lze považovat za kompletní, uvažujeme-li pouze ekvivalenční třídy o mohutnosti $m = 2^n \; \forall n \in \mathbb{N}$. Není-li mohutnost třídy mocninou 2 s přirozeným mocnitelem, pak její dvojkový logaritmus není celé číslo a nedává smysl jako počet kódovaných bitů. Existuje více možností, jak s takovým případem naložit:

1. Redukovat mohutnost ekvivalenční třídy na nejbližší nižší mocninu 2. Je vhodné odstranit členy, které se v průměru nejméně přirozeně vyskytují v krycích objektech daného typu, z důvodu maximalizace kapacity krycích objektů.
2. Deklarovat, že členy přebývající přes nejvyšší mocninu 2 nižší nebo rovnou mohutnosti m, tj. ty s indexy $\forall i \geqslant 2^{\lfloor \log_2 m \rfloor}$, nekódují žádnou validní posloupnost bitů. Při vkládání utajených dat pak budou vždy zaměněny za jiný člen; při extrakci vyvolají chybu.

3. Kódovat jednu ukrývanou posloupnost bitů více členy ekvivalenční třídy. Přebývající členy můžeme upotřebit například s využitím modulární aritmetiky: každý i-tý člen ekvivalenční třídy o mohutnosti m kóduje dvojkovou reprezentaci $i \mod 2^{\lfloor \log_2 m \rfloor}$.

4. Kódovat členy ekvivalenční třídy různě dlouhé řetězce bitů; například rozšířit délku ukrývané posloupnosti bitů u přebývajících členů následujícím způsobem: každý i-tý člen ekvivalenční třídy o mohutnosti m kóduje dvojkovou reprezentaci i na $\lfloor \log_2 m \rfloor$ bitech $\forall i < 2^{\lfloor \log_2 m \rfloor}$ a na $\lfloor \log_2 m \rfloor + 1$ bitech $\forall i \geqslant 2^{\lfloor \log_2 m \rfloor}$.

Příklad: Mějme ekvivalenční třídu o mohutnosti $m = 5$. Nejnižší nižší nebo rovná mocnina 2 je $2^{\lfloor \log_2 m \rfloor} = 4$. Členy s indexem menším než 4 tedy kódují binární reprezentaci svého indexu na $log_2 4 = 2$ bitech, tj. členy 0, 1, 2 a 3 kóduji posloupnosti bitů `00`, `01`, `10` a `11` v tomto pořadí. Se členem o indexu 4 můžeme, ve vztahu ke čtyřem výše uvedeným možnostem, naložit některým z následujících způsobů:

1. Odstranit ho z ekvivalenční třídy, případně dát přednost jinému členu s nižší pravděpodobností výskytu.

2. Zaměnit všechny jeho výskyty za příslušné ekvivalenty s indexy $i \in 0, 1, 2, 3$.

3. Kódovat jím binární reprezentaci $4 \mod 4 = 0$, tj. `00`.

4. Kódovat jím jeho binární reprezentaci na rozšířeném počtu bitů oproti předcházejícím členům, dostáváme tedy `100`.

Všimněme si jevu, který vznikne užitím poslední z možností. Předpokládejme, že jde o jedinou ekvivalenční třídu definovanou pro daný stegosystém; například řetězec bitů `10010100` pak zakódujeme posloupností tří členů ekvivalenční třídy $(4, 2, 4)$, ale na stejně dlouhý řetězec bitů `00101011` již potřebujeme posloupnost čtyř členů $(0, 2, 2, 3)$. Z toho vyplývá, že krycí objekty tohoto stegosystému mají variabilní kapacitu závislou na ukrývaných datech. V důsledku toho je třeba mluvit o minimální a maximální, případně průměrné kapacitě krycích objektů.

Objem výsledných stegogramů

Za použití substitučních metod lze dosáhnout výsledného stegogramu stejného objemu jako má vstupní krycí objekt, jsou-li použité sekvence bitů v krycím objektu stejné délky, jako ekvivalentní sekvence bitů, za něž jsou nahrazeny. Byť se teoreticky ani prakticky nejedná o vlastnost všech stegosystémů založených na substitučních metodách, je natolik typická, že ji některé zdroje ([13], [8]) zahrnují do samotné definice substitučních metod.

Příklad substituční metody: *LSB Metoda*

LSB metoda [24] patří mezi nejznámější steganografické techniky. Spočívá v záměně nejméně významných bitů (*least significant bits*) krycích dat za bity ukrývaných dat. Tato metoda typicky nachází uplatnění, je-li krycím objektem vzorkované audio či obrazová data v nekomprimované či bezeztrátově komprimované formě, a využívá výše zmíněné nedokonalosti lidských smyslů.

Například v případě obrázku reprezentovaných v prostorové doméně (tj. formáty *BMP*, *PNG*, ...) s barevnou hloubkou *True Color* je každý pixel reprezentovaný uspořádanou

trojicí 8bitových intenzit jednotlivých barevných složek (červené, zelené, modré), popřípadě čtveřicí, kde posledním, rovněž 8bitovým členem je míra neprůhlednosti (alfa-kanál). V tomto případě je možné využít každý bajt dekomprimovaných užitečných dat k zakódování 1 bitu ukrývaných dat, čímž získáme kapacitu rovnu $\frac{1}{8} = 12.5\%$ objemu užitečných dat. Přitom dojde u každého změněného barevného/alfa kanálu ke zvětšení či zmenšení o 1, tj. o $\frac{1}{2^8} \approx 0.4\%$ rozsahu, což lidský zrak nepostřehne.

Příklad substituční metody: *Lingvistické a formátové metody*

Jak bylo již několikrát zmíněno, krycím objektem může být i literární dílo. Na to lze pohlížet buď jako na nějakou strukturu jazykových konstrukcí, například posloupnost slov, nebo jako na sekvenci znaků nějaké znakové sady.

V prvním z uvedených případů lze využít některé z lingvistických metod [23], které staví na vhodných vlastnostech jazykových jevů. Lze například sestavit ekvivalenční třídy slov stejného významu (synonym; zkratek/akronymů a jejich expanzí; alternativních spellingů slov v různých variantách daného jazyka; ...). A sekvenčně kódovat do jejich výskytů dílčí řetězce bitů ukrývaných dat.

V druhém z uvedených případů lze použít některou z formátových metod [23], jež využívají vhodných vizuálních vlastností znaků, či jejich posloupností. Analogicky k lingvistickým metodám lze například i v tomto případě sestavit ekvivalenční třídy, jejichž členy jsou však posloupnosti znaků podobné grafické podoby. Například středník (*question mark*, `U+003B`) a řecký otazník (*greek question mark*, `U+037E`) ve znakové sadě *Unicode* jsou graficky téměř nerozeznatelné. O něco perceptibilnější je například záměna elipsy a posloupnosti tří teček. [23]

2.6.3 Generační metody

Generační metody (*generation-based methods* [8] nebo též *propagation steganography* [13]) jsou speciální kategorií steganografických metod zásadně se lišící od dvou předchozích tím, že neukrývají data do existujících krycích objektů, nýbrž do falešných, dynamicky generovaných. Jediným vstupem do generačního vkládacího algoritmu jsou data k ukrytí; výstupní stegogram je validní instancí nějakého datového formátu, jejíž obsah je však bezvýznamný.

Evidentní výhodou generačních metod je, že si dokáží krycí data „ušít na míru“ tak, aby byla co nejmenšího objemu, a přitom měla co největší kapacitu, a aby ukrytá data ve výsledném stegogramu byla co nejhůře detekovatelná.

Nevýhodou generačních metod je, že nedovolují využít existujících, smysluplných krycích dat, čímž vytvářejí přebytečnou režii ve veřejném komunikačním kanálu. Navíc je poměrně náročným úkolem algoritmická syntéza natolik autentických falešných krycích dat, aby pohled na komunikaci nevyvolal podezření u vnějšího pozorovatele.

Kapitola 3

Steganografie spustitelných souborů

Tato kapitola definuje pojem *spustitelný soubor* pro účely této práce, popisuje formát spustitelných souborů *ELF* a zmiňuje některá specifika rodiny instrukčních sad *x86*. Dále nastiňuje několik použitelných steganografických metod pro spustitelné soubory, včetně problémů které s sebou nesou, a mluví o jejich existujících implementacích.

3.1 Vymezení pojmu *spustitelný soubor*

Mluvíme-li obecně, spustitelný soubor (*executable file*) je jakýkoliv soubor, který obsahuje nějakým způsobem kódovaný *program*. Lze uvažovat dvě základní kategorie:

1. Zdrojové kódy psané v interpretovaných programovacích jazycích (např. *Python*, *JavaScript*, *PHP*, ...) a skripty sloužící k automatizaci systémových úloh (např. *bash-scripty* na unixových systémech, dávkové soubory v systému Windows). Ke svému spuštění vyžadují příslušný interpret, kterému jsou buď explicitně předány uživatelem, či samy obsahují informaci o svém interpretu (např. *shebang* na unixových systémech).

2. Kontejnery nesoucí strojový kód určený přímo pro procesor, případně přenositelný bajtkód pro běh na nějakém virtuálním stroji (např. *Java Virtual Machine*). Kromě samotné posloupnosti strojových instrukcí obyčejně obsahují různá metadata pro svoje běhové prostředí (tj. typicky operační systém či virtuální stroj) potřebná například ke korektnímu načtení programu do paměti, jeho inicializaci, jeho pohodlnému ladění a podobně.

Obecně se zbytek práce zabývá výlučně druhou ze zmíněných kategorií; konkrétně je vše ilustrováno na formátu spustitelných souborů *ELF* a rodině instrukčních sad *x86*, jež jsou popsány dále.

3.1.1 Formát ELF

ELF (*Executable and Linkable Format* [1] je formát spustitelných souborů, objektových souborů, a sdílených knihoven vyvinutý a publikovaný společností *Unix System Laboratories* jako součást specifikace *ABI (application binary interface)* pro čtvrté vydání operačního systému *Unix System V*. Setkáme se s ním na všech běžně užívaných systémech unixového

typu včetně *GNU/Linuxu*. Za zmínku stojí, že je formát navržen tak, aby byl snadno rozšířitelný a platformně nezávislý, tj. nesvázaný s konkrétním typem procesoru či instrukční sadou, čímž mimo jiné otevírá širší možnosti pro eventuální rozšíření této práce.

Struktura ELF souborů

ELF soubory se skládají ze čtyř základních částí:

- **ELF hlavička** (*ELF header*) povinně umístěná na začátku každého ELF souboru obsahuje, mimo jiné:
  - základní identifikační informace o souboru, tj. konstantní „magické bajty“ indikující, že jde o ELF soubor, bitovou šířku cílové architektury, použitou endianitu u vícebajtových položek atd.;
  - typ souboru (spustitelný, objektový, či knihovna);
  - identifikátor cílové architektury;
  - virtuální adresu vstupního bodu programu;
  - umístění dvou níže zmíněných tabulek v souboru a velikosti a počty jejich záznamů.
- **Tabulka hlaviček sekcí** (*section header table*) je pole struktur, jež popisují jednotlivé **sekce** obsažené v ELF souboru (popsány níže).
- **Tabulka programových hlaviček** (*program header table*) je pole struktur popisujících jednotlivé **segmenty** obsažené v ELF souboru (popsány níže).
- **Užitečná data** (*payload*) tvoří obsah výše zmíněných sekcí a segmentů. Zde je třeba zdůraznit, že sekce a segmenty se překrývají – jedná se o dva komplementární pohledy na ta stejná data. Bližší popis následuje.

Sekce

Sekce (*sections*) jsou pohledem na užitečná data upotřebitelným především při linkování a relokaci, jejich přítomnost je však cenná například i při analýze a disassemblování spustitelných souboru, jelikož poskytují jemnější a specifičtější pohled na užitečná data než segmenty. Jedna hlavička sekce poskytuje informace zejména o:

- názvu dané sekce;
- typu sekce (informace s programem definovaným významem, tabulka symbolů, tabulka řetězců, relokační informace, ...);
- umístění sekce v souboru a její délce;
- virtuální adrese sekce (stává-li se při spouštění součástí paměťového obrazu programu).

Jeden ELF soubor může obsahovat teoreticky neomezený počet libovolných uživatelských sekcí; vedle nich specifikace formátu definuje tzv. *speciální sekce (special sections)* snadno rozpoznatelné podle vyhrazené počáteční tečky v názvu, jež drží podstatné informace o programu využívané systémem. Patří mezi ně například:

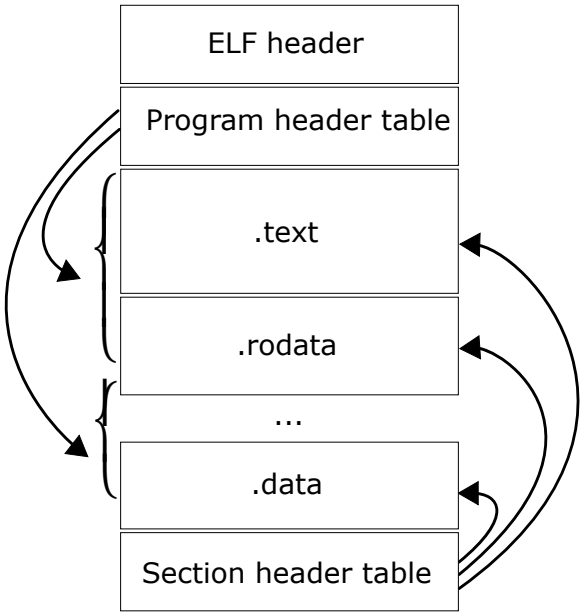
- sekce `.text` obsahující strojový kód (posloupnost instrukcí) programu;
- sekce `.data` obsahující inicializovaná data, jež po spuštění programu tvoří část jeho paměťového obrazu s přístupem ke čtení i zápisu (globální proměnné);
- sekce `.rodata` obsahující inicializovaná data, jež po spuštění programu tvoří část jeho paměťového obrazu s přístupem pouze ke čtení (konstanty);
- sekce `.strtab` obsahující tabulku řetězců;
- sekce `.symtab` obsahující tabulku symbolů;
- sekce `.rel` obsahující relokační informace.

Segmenty

Segmenty (*segments*) jsou alternativním pohledem na užitečná data využívaným při spouštění programu. Každý segment se fyzicky skládá z několika sekcí, tj. jeho obsah v užitečných datech je zároveň obsahem jedné či více sekcí. Výše zmíněné programové hlavičky obsahují zejména následující metadata:

- typ segmentu (data k načtení do paměti, informace o dynamickém linkování, ...);
- pozici segmentu v souboru a jeho virtuální adresu v paměti;
- délku segmentu v souboru a v paměti.

Obrázek 3.1: Struktura ELF souboru.



Steganografický potenciál ELF souborů

Položme si otázku, jak lze ELF soubor obecně použít jako krycí objekt. Možností je, jako vždy, teoreticky mnoho; jmenujme některé z nich:

- **Injekce krycích sekcí:** Jak je zmíněno výše v textu, každý ELF soubor smí obsahovat teoreticky neomezený počet uživatelských sekcí. Není tedy problém přidat jednu sekci navíc, jež bude sloužit pouze jako kontejner tajně přenášených dat. Samozřejmě je třeba myslet na všechny dříve zmíněné nevýhody *injekčních metod*.
- **Využití ladících informací:** ELF soubory mohou obsahovat ladící informace, např. ve formátu DWARF (*Debugging With Attributed Record Formats*) [7] původně vyvinutém přímo pro ELF. Předpokládáme-li, že bude daná instance spustitelného souboru jen spouštěna, nikoliv laděna, můžeme ladící data použít jako krycí.
- **Substituce strojového kódu**: Je faktem, že několik různých posloupností instrukcí může mít stejnou sémantiku, jejich záměna tudíž nezpůsobí změnu sémantiky programu. Lze tedy využít dříve zmíněné metody *ekvivalenčních tříd*. Zbytek práce se zaměřuje pouze na tento přístup.

3.1.2 Rodina instrukčních sad x86

x86 [19] je obecné označení pro rodinu zpětně kompatibilních instrukčních sad navazujících na instrukční sadu procesoru *Intel 8086* uvedeného na trh v roce 1978. Dnes je stále implementuje většina procesorů dominujících na trhu osobních počítačů a serverů. V dnešní době má smysl mluvit o dvou variantách x86:

- 32bitové **IA-32** (*Intel Architecture 32-bit*) někdy označované *i386* po prvním procesoru, jenž ji implementoval, tj. *Intel 80386* uvedeném na trh v roce 1985;
- 64bitové **x86-64**, někdy označované *AMD64*, která je 64bitovým rozšířením *IA-32* navrženým společností *AMD* poprvé implementovaným v procesorech *AMD Opteron* uvedených na trh v roce 2003.

Zbytek textu práce, především implementační část, demonstruje dále popsané postupy výlučně na této rodině instrukčních sad.

3.2 Použitelné steganografické metody

Jmenujme několik popsaných [4] steganografických metod využívajících posloupnost strojových instrukcí jako krycí objekt.

3.2.1 Permutace řetězů základních bloků

Základní blok (basic block) je posloupnost instrukcí, která splňuje následující podmínky:

1. Má pouze jeden vstupní bod (*entry point*), kterým je první instrukce bloku. Žádná z následujících instrukcí bloku nesmí ležet na cílové adrese jakékoliv skokové instrukce ležící kdekoliv v kódu (včetně bloku samotného).

2. Má pouze jeden výstupní bod (*exit point*), kterým je poslední instrukce bloku. Žádná z předchozích instrukcí bloku nesmí způsobit skok kamkoliv v kódu (včetně bloku samotného).

Některé základní bloky musí být umístěny za sebou v konkrétním pořadí, jelikož může docházet k propadům mezi nimi; výstupním bodem základního bloku totiž může být i některá z instrukcí **podmíněného** skoku, jejíž podmínka nemusí být vždy splněna. Jedna posloupnost všech sousedních bloků trpících touto vlastností se nazývá *řetěz (chain)* základních bloků.

Jelikož na uspořádání řetězů již nezáleží, mohou být mezi sebou libovolně permutovány. Toho lze využít ke steganografickému ukrytí dat – můžeme definovat ekvivalenční třídu, jejímiž členy jsou jednotlivé permutace řetězů. Každá z $n!$ permutací n řetězů pak reprezentuje jednu z možných posloupností $\log_2 n$ bitů.

Jako každá jiná metoda, ani tato se neobejde bez problémů, jež je potřeba řešit:

- Součástí stegosystému musí být jednoznačný, oběma utajeně komunikujícími stranami respektovaný, způsob indexace členů ekvivalenční třídy, tj. jednotlivých permutací řetězů, aby bylo jasné, jaké posloupnosti bitů kódují. V prvé řadě tedy musí být jednotně definováno uspořádání libovolné množiny řetězů. Jsou dvě možnosti:
  - Má-li příjemce stegogramu ve všech případech užití daného stegosystému k dispozici původní, nemodifikovaný krycí objekt, za uspořádanou může být považována permutace v něm.
  - Nemá-li příjemce stegogramu k dispozici původní krycí objekt (typičtější), je potřeba definovat uspořádání libovolné množiny řetězů obecně. Na mysl přichází například aplikace vhodné hashovací funkce na jednotlivé řetězy a porovnání jejích výstupů.
- Ne každý z řetězů je nutně unikátní, ergo ne každá permutace řetězů je nutně unikátní. Každou množinu všech neunikátních permutací řetězů je třeba redukovat pouze na jeden člen ekvivalenční třídy.
- Přemístění kódu ve spustitelném souboru samozřejmě způsobí změnu jeho adresy v paměťovém obrazu programu. To je potřeba reflektovat nalezením všech instrukcí skoku a volání, které míří dovnitř přemístěného řetězu, a změnou jejich cílových adres.

3.2.2 Permutace nezávislých instrukcí

Permutovány mohou být i některé instrukce uvnitř výše zmíněných základních bloků, je však třeba dát pozor na závislosti mezi nimi.

Je tedy nutné:

1. sestrojit graf závislostí instrukcí – opět jednotným způsobem nezávislým na vstupní permutaci;
2. vygenerovat, například s použitím *metody větví a mezí*, všechny možné permutace;
3. vybrat permutaci kódující příslušnou posloupnost bitů a zaměnit za ni původní kód.

3.2.3 Substituce instrukcí

Má-li několik instrukcí v daném kontextu stejnou sémantiku, je možné je libovolně zaměňovat bez změny sémantiky krycího strojového kódu jako celku. Každá taková množina instrukcí je tedy *ekvivalenční třídou* a můžeme se znovu uchýlit k postupům popsaným v předchozí kapitole práce. Před implementací stegosystému založeného na této metodě je opět třeba zamyslet se nad několika potenciálními komplikacemi.

Různá délka instrukcí

Různé instrukce mohou zabírat různé počty bajtů (viz popis instrukční sady x86 výše). Chceme-li připustit ekvivalenční třídy, jejichž členy jsou instrukce různých délek, musíme čelit dodatečným komplikacím. Záměna instrukce za jinou instrukci odlišné délky způsobí:

1. změnu velikosti obalující sekce a segmentu a posun těch následujících, je tedy třeba opravit příslušná metadata v ELF souboru;
2. posun všech následujících instrukcí o diferenci délek zaměňovaných instrukcí, což pravděpodobně zneplatní destinace některých instrukcí skoku a volání v kódu – ty je potřeba vyhledat a opravit;
3. odlišnou velikost výsledného stegogramu oproti původnímu krycímu objektu (nepadne-li součet všech způsobených posunů náhodou na 0), což potenciálně zvýší jeho podezřelost.

Závislost na kontextu

Chování některých instrukcí je závislé na kontextu, ve kterém se nacházejí. Typicky jde o instrukce, které testují některé z příznaků z registru *EFLAGS/RFLAGS*. Jako první přicházejí na mysl instrukce podmíněných skoků, ale existují i jiné příklady, třeba `ADC` (*add with carry*) a `SBB` (*subtract with borrow*), které k cílovému operandu přičítají/odečítají zdrojový operand sečtený s hodnotou příznaku přenosu (*carry flag*).

Chceme-li připustit ekvivalenční třídy instrukcí, jež různě využívají příznakový registr, musíme si pro každého nalezeného kandidáta na záměnu a jeho ekvivalent vždy položit dva dotazy:

1. **Zohledňují příznakový registr různým způsobem?** Například instrukce `ADD` a `ADC` jsou ekvivalentní pouze, je-li zaručeno, že v době jejich vykonání bude příznak přenosu vždy nulový. V opačném případě není možné instrukce zaměnit.
2. **Ovlivňují příznakový registr různým způsobem?** Například instrukce `ADD` s okamžitým operandem (*immediate operand*) je ekvivalentní instrukci `SUB` s okamžitým operandem, znegujeme-li okamžitý operand při jejich záměně. Jelikož však jinak nastavují příznaky přenosu, polovičního přenosu (*adjust flag*) a přetečení (*overflow flag*), je bezpečné je zaměnit pouze v případě, že jsou jmenované příznaky následujícím kódem ignorovány – tedy nejsou testovány některou následujících instrukcí – dokud nedojde k jejich modifikaci následujícími instrukcemi.

3.3 Existující implementovaná řešení

Za zmínku stojí aplikace *Hydan*[9]. Jedná se o implementaci metody substituce instrukcí nad instrukční sadou *IA-32* s podporou několika formátů spustitelných souborů (např. výše

rozebraného *ELF*, ale též *PE/COFF* používaného na operačním systému *Windows*). Implementačním jazykem aplikace je C; pro disassembling krycích programů používá knihovnu *libdisasm*[1].

Zdrojový kód aplikace je volně dostupný na jejích webových stránkách[2]. Je stručně ale přehledně zdokumentovaný a byl cenným zdrojem inspirace při implementaci vlastního řešení popsaného v následující kapitole.

[1] http://bastard.sourceforge.net/libdisasm.html
[2] http://www.crazyboy.com/hydan/

Kapitola 4

Popis vlastní implementace

Tato kapitola dokumentuje ukázkovou konzolovou aplikaci prakticky demonstrující steganografii spustitelných souborů, jež je součástí této práce. Jsou zde zmíněny jak teoretické principy, na kterých aplikace staví, tak i různé implementační detaily.

4.1 Použitá metoda

Demonstrační aplikace je postavena na metodě substituce instrukcí popsané v předchozí kapitole.

4.1.1 Vlastnosti ekvivalenčních tříd

Kolekce ekvivalenčních tříd instrukcí, již implementace využívá ve svojí současné verzi, je snadno rozšiřitelná vhodnou modifikací konfiguračního souboru `eq-classes.json` popsaného v následujících sekcích kapitoly. Z povahy implementace však mají všechny – nyní definované i eventuálně přibyvší – ekvivalenční třídy instrukcí následující vlastnosti:

- **Členy stejných délek:** Členské instrukce jedné ekvivalenční třídy jsou vždy stejných bajtových délek. Při substituci instrukce o délce n tedy stačí přepsat n bajtů od pozice instrukce v krycím objektu kódem příslušné ekvivalentní instrukce – není třeba se zabývat rozložením okolního kódu.
- **Libovolná mohutnost:** Počty členů ekvivalenčních tříd nejsou nutně mocniny 2. Problém neceločíselné entropie zaměňovaných posloupností bitů je řešen metodou variabilního počtu kódovaných bitů (popsanou v části sekce 2.6.2), jež se zabývá konceptem ekvivalenčních tříd.
- **Závislost na kontextu:** Je brán ohled na případný rozdílný vliv členských instrukcí na hodnoty v příznakovém registru (popsaný v sekci 3.2.3). Jejich záměna je dovolena pouze v případě, že nezmění sémantiku následujícího kódu. Konkrétní algoritmus je popsán dále v této kapitole.

4.1.2 Užité druhy ekvivalenčních tříd

Každý z principů uvedených v následujících odstavcích je implementován alespoň jednou (zpravidla větším počtem) z ekvivalenčních tříd obsažených v konfiguračním souboru

`eq-classes.json` popsaném dále v textu práce; některé ekvivalenční třídy principy kombinují. Seznam není teoreticky vyčerpávající a jeho rozšíření je námětem na vylepšení demonstrační aplikace.

Přičítání negace vs. odečítání

Tento princip byl již jednou zmíněn v předchozí kapitole: přičtení (`ADD`) a odečtení (`SUB`) okamžitého operandu je ekvivalentní, znegujeme-li okamžitý operand v jednom z případů, jelikož $a - b = a + (-b)$. Jak bylo taktéž zmíněno, je třeba zajistit, že rozdílná modifikace příznaků `CF`, `OF` a `AF` těmito dvěma instrukcemi nezmění sémantiku následujícího kódu.

V současné verzi demonstrační aplikace je záměrně vynechána kontrola testování příznaku `AF` následujícími instrukcemi, jelikož jeho primárním uživatelem jsou zřídkakdy používané *BCD* (*binary-coded decimal*) operace.

Záměna operandů a směrového bitu

Mějme libovolnou instrukci, kde oba operandy jsou registry (`mod` bity *ModR/M* bajtu instrukce jsou `11`[22]). I v takovém případě rodina instrukčních sad *x86* umožňuje nastavovat hodnotu *směrového bitu* (*direction bit*, součást operačního kódu instrukce), jenž určuje, který z operandů je zdrojový a který cílový. Prohodíme-li operandy a zároveň invertujeme směrový bit, změní se zápis instrukce, nicméně nedojde ke změně její sémantiky.

Odečtení vs. bitová exkluzivní disjunkce sebe sama

Existuje více způsobů, jak vynulovat registr, například `MOV <reg>, 0`, `SUB <reg>, <reg>`, či `XOR <reg>, <reg>`. Poslední dvě z uvedených možností mají naštěstí stejnou bajtovou délku, lze tedy jednu snadno substituovat za druhou. Příznakový registr je instrukcemi modifikován stejným způsobem až na příznak `AF`, který je však současnou verzí demonstrační aplikace ignorován (viz výše).

Ekvivalentní bitové operace nad sebou samým

Instrukce `TEST` má dva zdrojové operandy, nad kterými provede operaci bitového logického součinu a příslušným způsobem nastaví příznaky v příznakovém registru – stejně, jako by to udělala instrukce `AND`, která by však navíc uchovala výsledek v cílovém operandu.

Zaměřme se na případ, kdy jsou zdrojové operandy instrukce `TEST` totožné (tj. konkrétně jde o ten samý registr, jelikož dva paměťové operandy instrukční sada nepřipouští). V takovém případě je použití instrukcí `TEST` a `AND` ekvivalentní. Operace bitového logického součinu je idempotentní[1], ergo cílový operand instrukce `AND` bude mít po jejím provedení stejnou hodnotu jako před ním.

Třetím kandidátem na člena této ekvivalenční třídy je instrukce `OR`, která je rovněž idempotentní a modifikuje příznakový registr stejným způsobem.

4.2 Použité technologie

Uveďme technologie použité pro vývoj aplikace a zároveň potřebné pro úspěšný běh aplikace.

[1] Aplikací operace na dvě totožné vstupní hodnoty získáváme na výstupu opět vstupní hodnotu.

4.2.1 Python 3

Python je vysokoúrovňový interpretovaný programovací jazyk kladoucí důraz na čitelnost a jednoduchost kódu. Jeho velkou předností je rozsáhlá standardní knihovna, díky které je programátor ušetřen řešení nízkoúrovňových problémů bez závislosti na balíčcích od třetích stran. Pro správný běh demonstrační aplikace je třeba *Python 3.6* nebo vyšší.

4.2.2 GNU Binutils a nástroj `objdump`

GNU Binutils je sada nástrojů vyvinutých primárně pro *GNU/Linux* sloužící k různé manipulaci s objektovými a spustitelnými soubory. Pro tuto práci je důležitý nástroj `objdump`, který demonstrační aplikace používá k disasemblování strojového kódu a zjišťování některých metadat zpracovávaných ELF souborů.

4.3 Popis implementace aplikace

Demonstrační konzolová aplikace se spouští následovně:

```
./thesis --<režim> --cover <cesta_ke_krycímu_objektu_či_stegogramu>
```

Jak naznačuje příklad spuštění výše, aplikace lze spustit v několika režimech:

- Režim **disassemble** pouze disasembluje strojový kód z krycího objektu a vypíše základní informace o každé instrukci (hexadecimální a binární reprezentaci kódu, mnemoniku, operandy a pozici v souboru) na standardní výstup. Tento režim přijde vhod především při vývoji aplikace.
- Režim **analyze** pouze provede dále popsanou analýzu krycího objektu a vypíše jeho metriky – minimální a maximální kapacitu a počty – na standardní výstup.
- Režim **embed** provede analýzu krycího objektu a ukryje do něj maximální možný počet bitů dat ze standardního vstupu. Je-li naplněna kapacita krycího objektu před zakódováním celé délky ukrývaných dat, uživatel je informován skrze standardní chybový výstup. Výsledný stegogram je odeslaný na standardní výstup (zdrojový soubor s krycím objektem není modifikován).
- Režim **extract** provede analýzu stegogramu, extrahuje z něj ukrytá data a odešle je na standardní výstup.

Aplikaci lze spustit v explicitním – *verbose* – módu (pomocí stejnojmenného přepínače), který v případě některých z výše uvedených režimů poskytuje dodatečné ladící informace tisknuté na standardní chybový výstup. Každý z přepínačů má i svoji zkrácenou formu (např. `-e` pro `--embed`, `-x` pro `--extract`, ...). Kompletní přehled možností spouštění aplikace je k dispozici pod přepínačem `--help`.

Následuje popis implementace jednotlivých částí programu.

4.3.1 Disassembling strojového kódu

Disasemblování strojového kódu v krycích objektech je delegováno na výše uvedený nástroj `objdump`. Ten je jako subproces aplikace spuštěn nad krycím ELF souborem s příslušnými argumenty, aby na standardní výstup vytiskl:

1. člověkem čitelný výpis informací z hlavičky dříve zmíněné sekce `.text`, zejména její umístění v souboru a virtuální adresu;
2. disasemblovaný obsah sekce `.text`, tj. člověkem čitelný výpis jednotlivých strojových instrukcí programu zahrnující jejich virtuální adresy, hexadecimální bajtové reprezentace a zápisy v jazyku symbolických instrukcí.

```
objdump -hdj .text <cesta_ke_krycímu_objektu_či_stegogramu>
```

Standardní výstup subprocesu je odchycen a řádek po řádku rozparsován prostřednictvím regulárních výrazů. Výsledkem je lineární seznam instancí třídy `Instruction` – reprezentujících jednotlivé strojové instrukce – s následujícími datovými členy:

- `offset` – pozice instrukce v souboru (získaná přičtením rozdílu pozice a adresy sekce `.text` k adrese instrukce);
- `code` – strojový kód instrukce jako datový typ *bytes* (imutabilní pole bezznaménkových celých čísel);
- `mnemonic` – řetězec s mnemonikou instrukce (`"mov"`, `"push"`, ...);
- `operands` – operandy instrukce jako člověkem čitelný řetězec využívaný čistě pro informační výpisy.

Výše popsaný postup je úlohou iniciálizátoru třídy `Disassembly`, jejíž instance obaluje seznam vyparsovaných instrukcí a poskytuje rozhraní pro iteraci nad nimi.

4.3.2 Konfigurační soubory

Před provedením analýzy získané posloupnosti instrukcí je třeba načíst příslušné informace ze dvou konfiguračních souborů a transformovat je na vhodné datové struktury:

`config/eq-classes.json`

Tento konfigurační soubor ve formátu *JSON* obsahuje kolekci samotných ekvivalenčních tříd. Každá ekvivalenční třída je reprezentovaná objektem s následujícími vlastnostmi:

- `label` – Člověkem čitelný popis ekvivalenční třídy používaný při statistickém výpisu v režimu *analyze*.
- `members` – Kolekce jednotlivých členů ekvivalenční třídy citlivá na pořadí (členy kódují svoje indexy, jak je popsáno v 2.6.2). Každý člen je reprezentován objektem s následujícími vlastnostmi:
  - `match` – Regulární výraz popisující člověkem čitelnou binární reprezentaci kódu instrukce s jednotlivými bitovými oktety v *MSB-first*[2] pořádí oddělenými mezerami. Může obsahovat *pojmenované skupiny* (*named groups*) [12] sloužící k odchycení variabilních složek instrukce, jako jsou hodnoty okamžitých operandů, nebo bitové triplety `reg` a `rm` kódující zdrojový a cílový registr operace. Využívá se výlučně ve fázi analýzy krycích dat pro rozpoznání, zda je evaluovaná instrukce instancí testovaného člena ekvivalenční třídy.

[2] Od nejvýznamnějšího bitu k nejméně významnému zleva doprava, např. desítková hodnota 42 je zapsána jako `00101010`.

  - `generate` – Formátovací řetězec definující kód instrukce ve stejném formátu jako výše zmíněný regulární výraz. Kromě nul, jedniček a mezer může obsahovat proměnné ohraničené složenými závorkami, jež by měly odpovídat názvům výše uvedených pojmenovaných skupin.
  - `negated` – Kolekce názvů pojmenovaných skupin, jejichž hodnoty jsou považovány za „znegované“. Při substituci instrukcí je třeba provést negaci (ve smyslu dvojkového doplňku) všech hodnot odchycených pojmenovaných skupin, které se nachází v kolekci `negated` nahrazovaného členu a **ne**nachází v téže kolekci nahrazujícího členu nebo naopak. Příkladem je dříve zmíněná ekvivalence přičítání a odečítání, je-li v jednom z případů operand znegovaný – při nahrazení přičítání odčítáním, či naopak, je třeba znegovat operand.

- `flags` – Kolekce identifikátorů příznaků z příznakového registru (např. `"CF"`, `"OF"`, ...), které jsou rozdílně ovlivněny jinak ekvivalentními členskými instrukcemi ekvivalenční třídy.

Soubor je ručně vytvářený. Proto je, z důvodu lepší orientace v něm, jeho formát rozšířen o validitu jednořádkových komentářů ve stylu jazyka C, přestože standardní *JSON* [14] je nedovoluje (před parsováním souboru jsou nalezeny a zahozeny). Toho je využito k označení jednotlivých členů ekvivalenčních tříd jejich mnemonikami a operandy.

`config/flag-usages.json`

Účel tohoto konfiguračního souboru je asociace instrukcí s informacemi o jejich interakci s příznakovým registrem. Konkrétně obsahuje mapu mnemonik instrukcí na objekty s následujícími volitelnými vlastnostmi:

- `tested` – Kolekce identifikátorů příznaků (`"CF"`, `"OF"`, ...), jež jsou testovány instrukcemi s danou mnemonikou. Absence vlastnosti značí, že instrukce s danou mnemonikou netestuje žádné příznaky.

- `modified` – Kolekce identifikátorů příznaků, jež jsou modifikovány instrukcemi s danou mnemonikou. Absence vlastnosti značí, že instrukce s danou mnemonikou nemodifikuje žádné příznaky.

- `jump` – Booleovská hodnota říkající, zda se jedná o skokovou instrukci. Absence vlastnosti je ekvivalentní hodnotě `false`.

Na rozdíl od `eq-classes.json` je tento konfigurační soubor automaticky generován z volně dostupné, strojově čitelné reference instrukcí *x86* ve formátu XML [18] pomocí samostatného skriptu (`misc/flag-usage-gen.py`).

4.3.3 Analýza krycích objektů

V analytickém, vkládacím i extrakčním režimu aplikace probíhá totožná analýza krycího objektu.

1. Je instanciována třída `StegoProcessor`, která řídí proces analýzy krycích dat. V jejím inicializátoru probíhají následující kroky.

2. Je vytvořena instance třídy `EqClassList` reprezentující kolekci všech ekvivalenčních tříd a poskytující iterační rozhraní nad nimi. Ta načítá a parsuje výše zdokumentovaný konfigurační soubor a dává vzniknout seznamu instancí třídy `EqClass` (jednotlivé ekvivalenční třídy) a jejich dceřiným seznamům instancí třídy `EqClassMember` (jednotlivé členy svých ekvivalenčních tříd).

3. Je vytvořena instance třídy `FlagUsageList`, která načítá a parsuje obsah výše popsaného `flag-usages.json` a implementuje funkcionalitu užitečnou při kontrole kontextu instrukcí, jež je popsána dále.

4. Je iterováno nad instancí dříve zmíněné třídy `Disassembly` a jsou postupně izolovány pouze instrukce, které jsou validními výskyty členů ekvivalenčních instrukcí, ergo schopné kódovat (v případě čistého krycího objektu) či již kódující (v případě dříve vytvořeného stegogramu) utajená data. Tento proces je detailněji popsán níže.

Po dokončení analýzy krycích dat poskytuje instance třídy `StegoProcessor` rozhraní pro:

- zjištění statistik o krycím objektu – viz režim *analyze* uvedený výše;
- vkládání utajených dat do izolovaných použitelných instrukcí;
- extrakci utajených dat zakódovaných v izolovaných instrukcích.

Izolace použitelných instrukcí

Výběr instrukcí použitelných jako krycí data je realizováno iterací nad všemi instrukcemi získanými z krycího objektu a jejich postupným testováním. Aby byla instrukce označena za použitelnou, musí splňovat tyto podmínky:

- **Je výskytem některého z členů nějaké ekvivalenční třídy.** Zjištění tohoto faktu je delegováno na třídu `EqClassList`, která dílčí úlohy předává dál: Zda testovaná instrukce odpovídá nějakému členu ekvivalenční třídy, umí rozhodnout sama třída `EqClassMember`. Kód instrukce je srovnán s regulárním výrazem popisujícím kód daného členu (viz 4.3.2), poté co je převeden na stejnou reprezentaci. Dopadne-li srovnání kladně, jsou odchyceny pojmenované skupiny (opět viz 4.3.2) a vrácena vhodná datová struktura skrz `EqClass` a `EqClassList`, které k ní přibalují potřebné informace navíc. Výsledkem je instance třídy `EqClassMatch`, jejíž popis je k nalezení níže.
- **Substituce za některý z jejích ekvivalentů nezmění sémantiku následujícího kódu.** Jsou dvě možnosti:
  - Členy dané ekvivalenční třídy, s jedním z nichž instrukce koresponduje, nemají rozdílný vliv na příznakový registr.
  - Příznaky, jež členy dané ekvivalenční třídy modifikují rozdílně, nejsou testovány dříve, než je přepíší následující instrukce. Algoritmus zjištění, zda toto tvrzení platí pro danou instrukci, je ilustrován následujícím pseudokódem:

```
Function jePouzitelna(kontrolovanaInstrukce, ekvivalencniTrida):
    zbyvajiciPriznaky ← ekvivalencniTrida.rozdilneModifikovanePriznaky();
    aktualniInstrukce ← kontrolovanaInstrukce;
    while |zbyvajiciPriznaky| > 0 ∧ aktualniInstrukce.maNaslednika() do
        aktualniInstrukce ← aktualniInstrukce.naslednik();
        if aktualniInstrukce.jeSkokova() ∨
         aktualniInstrukce.testujeNektere(zbyvajiciPriznaky) then
            return false;
        end
        zbyvajiciPriznaky ← zbyvajiciPriznaky
         −aktualniInstrukce.modifikovanePriznaky()
    end
    return true;
```

Výsledná datová struktura

Výsledkem izolační procedury je seznam instancí výše zmíněné třídy `EqClassMatch`, která reprezentuje jeden konkrétní výskyt členu ekvivalenční třídy v krycích datech a sdružuje všechny informace potřebné k následnému vkládání či extrakci utajených bitů do/z instrukce. Jejími datovými členy jsou:

- `offset` – pozice instrukce v krycím souboru;
- `eq_class` – ekvivalenční třída (instance `EqClass`);
- `member_index` – index odpovídajícího členu ekvivalenční třídy;
- `groups` – objekt slovníkového typu mapující názvy odchycených pojmenovaných skupin na instance třídy `CapturedGroup`, jejímiž datovými členy jsou:
  - `payload` – řetězec bitů odchycený danou pojmenovanou skupinou;
  - `negated` – booleovská hodnota značící, zda je odchycený řetězec bitů považován za „znegovaný“ (viz 4.3.2).

4.3.4 Vkládání a extrakce utajených dat

Základem vkládání i extrakce utajených dat je iterace nad seznamem instancí třídy `EqClassMatch`, jenž je výstupem výše popsané analýzy.

Vkládání

1. Libovolná data k zakódování jsou přečtena ze standardního vstupu.
2. Načtené pole bytů je konvertováno na seznam bitů (hodnot typu `int` $\in \{0, 1\}$) s *LSB-first*[3] řazením.
3. Je zjištěna délka seznamu ukrývaných bitů a seznam je prefixován její 24bitovou *LSB-first little-endian* reprezentací.

[3] Každá osmice bitů začíná nejméně signifikantním bitem a končí nejsignifikantnějším.

4. Úplná binární data krycího objektu jsou načtena do paměti (nezávisle nad jejich předchozím zpracování pomocí `objdump`u) jako mutabilní `bytearray`.

5. Iteruje se nad seznamem instancí třídy `EqClassMatch`. Každá iterace se skládá z následujících kroků:

   (a) Kódovacímu rozhraní uchované instance třídy `EqClass` jsou předána data krycího objektu, zbývající bity ukrývaných dat k zakódování a slovník odchycených skupin.

   (b) Z levého konce seznamu ukrývaných bitů je useknut vhodný počet bitů a dekódován na index členu ekvivalenční třídy.

   (c) Na instanci třídy `EqClassMember` pod daným indexem je delegováno vygenerování a vrácení kódu instrukce z dříve popsaného generačního formátovacího řetězce a předaného slovníku odchycených skupin.

   (d) Kód nově vygenerované instrukce nahrazuje kód nahrazované instrukce v datech krycího objektu.

6. Modifikovaná data krycího objektu jsou, jakožto výsledný stegogram, odeslána na standardní výstup.

7. Nepodařilo-li se vložit všechny bity ukrývaných dat kvůli nedostatečné kapacitě krycího objektu, je vyvolána výjimka, jejíž odchycení v hlavním modulu způsobí příslušný informační výpis na standardní chybový výstup.

Extrakce

Extrakce je výrazně jednodušší proces. Opět iterujeme nad seznamem instancí třídy `EqClassMatch`; v každé iteraci je dekódovacímu rozhraní uchované instance třídy `EqClass` předán index rozpoznaného členu, který je následně vrácen binárně reprezentován na odpovídajícím počtu bitů.

Po získání prvních 24 bitů, jež kódují bitovou délku zbytku ukrytých dat, je známo kolik zbývajících bitů očekávat. V momentě, kdy je očekávaný počet naplněn, jsou osmice bitů agregovány do standardní bajtové reprezentace a odeslány na standardní výstup.

Kapitola 5

Vyhodnocení funkčnosti implementace

5.1 Testování aplikace

Testování funkčnosti demonstrační aplikace by mělo vést především k ověření následujících tvrzení:

1. **Fungují procesy vkládání a extrakce tajných dat.** Tj. data extrahovaná ze stegogramu přesně odpovídají datům, jež do něj byla vložena; nedošlo k žádným ztrátám, deformacím apod.

2. **Chování výstupních stegogramů je totožné s chováním vstupních krycích objektů.** Stegogramy jsou tedy stále funkční spustitelné soubory, jež pro stejné vstupy produkují stejné výstupy, jako jejich steganograficky neposkvrněné originály.

Test používaný při vývoji aplikace je realizován jednoduchým shellovým skriptem nacházejícím se v přibaleném *Makefilu* pod cílem `test`. Jako testovací krycí objekt je použit program `ls`, jelikož se nachází na každém běžném unixovém systému (není ho tedy třeba přibalovat) a produkuje snadno porovnatelný výstup. Úspěšně proběhlý test vypadá následovně:

```
1 ./thesis -ec ‘which ls --skip-alias‘ < misc/test-message.txt > test-output
2 ./thesis -xc test-output > test-output.txt
3 diff misc/test-message.txt test-output.txt
4 chmod +x test-output
5 ./test-output / > test-execution.txt
6 ‘which ls --skip-alias‘ / | diff - test-execution.txt
7 rm -f test-output test-output.txt test-execution.txt
```

Stručně slovně popsáno: tajná textová zpráva je ukryta do programu `ls` dynamicky nalezeného pomocí příkazu `which` (na různých systémech se totiž může nacházet v různých adresářích, např. `/bin` či `/usr/bin`); výsledný stegogram je zapsán do souboru. Textová data ukrytá ve stegogramu jsou extrahována a porovnána s původní tajnou zprávou pomocí programu `diff`. Následně jsou stegogramu nastavena spouštěcí práva a je spuštěn (tak aby vypsal obsah kořenového adresáře, u něhož lze předpokládat konstantní obsah v průběhu skriptu). Jeho výstup je zachycen do souboru a porovnán s výstupem originálního `ls` spuštěného se stejným argumentem.

Vrátí-li kterýkoliv z příkazů chybový návratový kód, program `make` okamžitě ukončí provádění skriptu a signalizuje selhání chybovým výpisem. Standardní výstup neúspěšného `diff`u či chybový výstup naší aplikace mohou být následně nápomocny při diagnóze chyby.

Skript tedy kontroluje oba výše popsané aspekty funkčnosti demonstrační aplikace.

5.2 Profilování a optimalizace aplikace

Při testování demonstrační aplikace na objemnějších krycích objektech (stovky kB, jednotky MB) byla velmi zřetelná časová náročnost jejich analýzy. Vzniklé nepohodlí se rychle stalo motivací pro optimalizaci aplikace s pomocí profilování. Byl použit profiler *cProfile*[11] z následujících důvodů:

- jedná se o standardní modul Pythonu, čímž odpadá závislost na knihovnách třetích stran;
- je použitelný nejen jako samostatný program, ale i jako Pythonové API, profilování může tedy být realizováno přímo v rámci zdrojového kódu aplikace.

Byl tedy doimplementován přepínač `--profile`. Spuštění aplikace s ním způsobí aktivaci profileru pro hlavní část programu (disassembling a analýza krycího objektu; případné vkládání či extrakce dat v příslušných režimech) a uloží získanou statistiku v člověkem čitelné podobě do souboru, cesta k němuž je argumentem přepínače. Následně bylo do *Makefilu* přidáno pravidlo `profile`, které automatizuje spuštění aplikace ve vkládacím i extrakčním režimu a uložení profilovacích statistik do příznačně pojmenovaných textových souborů.

Získaná statistika odhalila, že kumulativní čas[1] strávený vykonáváním metody `code_binary` třídy `Instruction` tvoří $65,8\%$ doby běhu aplikace. Jedná se o metodu, jež provádí konverzi standardní bajtové reprezentace kódu instrukce (datový člen `code`, viz 4.3.1) na reprezentaci porovnatelnou s regulárními výrazy identifikujícími členy ekvivalenčních tříd (viz 4.3.2). Metoda je volána a znovu vykonána před každým porovnáním nějaké instrukce s některým z členů některé ekvivalenční třídy, přestože její návratová hodnota je, vzhledem k nemutovanosti instancí třídy `Instruction`, pokaždé stejná.

Na metodu byl aplikován dekorátor `lru_cache` ze standardního modulu `functools`, jenž memoizuje[2] návratovou hodnotu metody, která je tím pádem vygenerována pouze jednou pro každou instrukci. Následně bylo zopakováno profilování: kumulativní čas metody se snížil na $2,3\%$ původní hodnoty a tvoří nyní pouze $5,3\%$ doby běhu aplikace, jež se snížila na $29,2\%$ původní hodnoty.

5.3 Hodnocení aplikace dle steganografických kritérií

Je na místě předpokládat, že spustitelné soubory jsou poměrně nízkokapacitním typem krycího objektu; zaměřme se tedy na závěr na měření jejich kapacity a zamysleme se, jak ji maximalizovat.

Aby nebylo třeba provádět měření ručně, byl implementován jednoduchý analytický skript (`misc/cover-stats.py`), jenž importuje potřebné moduly demonstrační aplikace a

[1] Celkový čas strávený prováděním dané funkce a funkcí, které volá.

[2] Memoizace (*memoization*) je optimalizační technika založená na cachování návratových hodnot funkce a jejich vracení z mezipaměti pro opakovaná volání s identickými vstupy.

automaticky provádí disassembling a analýzu většího počtu testovacích krycích objektů najednou. Vstupem skriptu je cesta k adresáři obsahujícímu programy k analýze. Výstupem skriptu jsou data ve formátu *CSV*; každý záznam nese informace o jednom z krycích objektů: název, velikost v bajtech, minimální a maximální kapacitu v bitech, trvání analýzy v sekundách a počty výskytů členů jednotlivých ekvivalenčních tříd.

Skript lze spustit jako `make stats` – v takovém případě je proveden nad adresářem `execs`, ve kterém je připravena následující sada programů: `bc`, `cmp`, `espeak`, `gpg`, `make`, `openssl`, `sort`, `zip`. Od každého je přítomna vhodně označená 32bitová a 64bitová verze. Programy jsou vzaty z adresáře `/usr/bin` 32- a 64bitové verze operačního systému *Linux Mint 18.3*).

Poznámka: dodatečné zpracování naměřených údajů prezentované v následujících odstavcích bylo realizováno pomocí tabulkového procesoru *LibreOffice Calc*.

5.3.1 Vztah kapacity a velikosti krycího objektu

Následující tabulka dává do souvislosti naměřené bajtové velikosti krycích objektů (s) s jejich minimálními (c_{min}) a maximálními (c_{max}) bitovými kapacitami. Poslední dva sloupce obsahují minimální (r_{min}) a maximální (r_{max}) **relativní** kapacity (*encoding rates* [9]) krycích objektů, nebo-li poměry jejich absolutních kapacit ku jejich velikostem, vypočtené jako $\frac{c}{s \cdot 8}$ a zobrazené v procentech.

program	$s[B]$	$c_{min}[b]$	$c_{max}[b]$	$r_{min}[\%]$	$r_{max}[\%]$
`bc-32`	84048	3038	3463	0,45	0,52
`bc-64`	85232	1550	1787	0,23	0,26
`cmp-32`	46812	1984	2272	0,53	0,61
`cmp-64`	43696	704	827	0,20	0,24
`espeak-32`	18344	297	353	0,20	0,24
`espeak-64`	19464	188	211	0,12	0,14
`gpg-32`	1128632	55783	64123	0,62	0,71
`gpg-64`	1008632	28524	32458	0,35	0,40
`make-32`	212884	10583	12348	0,62	0,73
`make-64`	207528	4188	4727	0,25	0,28
`openssl-32`	617288	23542	27876	0,48	0,56
`openssl-64`	559072	12663	14389	0,28	0,32
`sort-32`	108704	5219	5797	0,60	0,67
`sort-64`	110040	2154	2353	0,24	0,27
`zip-32`	205764	8459	9731	0,51	0,59
`zip-64`	192520	3050	3591	0,20	0,23

Průměrná relativní kapacita krycích objektů v kontextu implementovaného stegosystému je pouze $0,58$ % v případě 32bitových programů a $0,27$ % v případě 64bitových. Vskutku se nejedná o vysoké hodnoty. Relativní kapacitě, již slibuje dříve zmíněný *Hydan* ($\frac{1}{110} \approx 0.90$ % [9]), odpovídají alespoň řádově.

Za povšimnutí stojí poměrově signifikantní rozdíl hodnot pro 32- a 64bitové programy. Ten je zřejmě způsoben faktem, že žádná z ekvivalenčních tříd nepopisuje instrukce specifické pro instrukční sadu *x86-64*. 64bitové programy sice obsahují instrukce kompatibilní se sadou *IA-32* (jejich kapacita tedy není nulová), logicky však menší relativní množství.

5.3.2 Vztah kapacity a časové složitosti analýzy

Instance některých ekvivalenčních tříd se v krycích objektech objevují jen zřídka, stále však signifikantně kontribuují k času spotřebovanému jejich analýzou. Lze předpokládat, že eliminace takových ekvivalenčních tříd povede ke zrychlení aplikace bez výrazného snížení kapacity krycích objektů. Experiment adresující tento předpoklad sestával z následujících kroků:

1. Analytický skript byl spuštěn nad původní kolekcí ekvivalenčních tříd.
2. Ekvivalenční třídy, jejichž počet výskytů tvořil 0.2 % z celkového počtu „zásahů“ nebo méně, byly v konfiguračním souboru dočasně zakomentovány.
3. Analytický skript byl spuštěn znovu, nad redukovanou kolekcí ekvivalenčních tříd.
4. Naměřené doby trvání analýzy (t_0 vs. t_1) a průměrné ($\frac{c_{min}+c_{max}}{2}$) kapacity krycích objektů (c_0 vs. c_1) byly poměřeny a zaneseny do následující tabulky:

program	$t_0[s]$	$c_0[b]$	$t_1[s]$	$c_1[b]$	$\frac{t_1}{t_0}[\%]$	$\frac{c_1}{c_0}[\%]$
`bc-32`	0,57	3250,5	0,38	3244,5	65,5	99,8
`bc-64`	0,56	1668,5	0,34	1665,5	61,1	99,8
`cmp-32`	0,37	2128	0,24	2121	64,4	99,7
`cmp-64`	0,31	765,5	0,19	761,5	61,3	99,5
`espeak-32`	0,06	325	0,04	325	65,0	100,0
`espeak-64`	0,06	199,5	0,04	199,5	63,3	100,0
`gpg-32`	55,11	59953	51,95	59871	94,3	99,9
`gpg-64`	10,15	30491	6,68	30440	65,8	99,8
`make-32`	2,16	11465,5	1,50	11426,5	69,4	99,7
`make-64`	1,70	4457,5	1,05	4438,5	61,7	99,6
`openssl-32`	9,65	25709	8,03	25637	83,3	99,7
`openssl-64`	3,98	13526	2,49	13510	62,4	99,9
`sort-32`	0,96	5508	0,64	5497	66,8	99,8
`sort-64`	0,86	2253,5	0,53	2246,5	61,8	99,7
`zip-32`	1,76	9095	1,20	9043	68,3	99,4
`zip-64`	1,38	3320,5	0,85	3289,5	61,6	99,1

Tabulka získaných hodnot evidentně potvrzuje výše popsanou hypotézu. Kapacita krycích objektů byla téměř zachována (průměrně snížena pouze o $0,29$ %), zatímco doba trvání analýzy klesla průměrně na $67,24$ % původní hodnoty.

Kapitola 6

Závěr

Aplikace demonstrující steganografii nad spustitelnými soubory jakožto krycími objekty byla úspěšně navržena, implementována, otestována, optimalizována, zhodnocena dle vhodných kritérií a do detailu zdokumentována v tomto textu. Vývoji aplikace předcházela důkladná rešerše a následná sumarizace existujících – již implementovaných i čistě teoretických – postupů. Nelze zanedbat ani množství vlastních úvah a experimentů, s nimiž je čtenář rovněž seznámen.

Navázat na vývoj aplikace lze více způsoby. Asi nejvíce se nabízí hlubší studium rodiny instrukčních sad *x86*, vygenerování co největšího počtu ekvivalenčních tříd instrukcí, a jejich evaluace za pomoci dalších statistických experimentů. O něco inovativnějším směrem vývoje by bylo rozšíření podpory různých instrukčních sad či formátů spustitelných souborů (po vzoru již několikrát zmíněné aplikace *Hydan*). Aplikaci by jistě též prospěla důkladnější optimalizace.

Srovnání s projekty podobného cíle bohužel nemůže být příliš květnaté, jedinou běžně dostupnou alternativou je totiž *Hydan*. Výsledek experimentu prezentovaný v předchozí kapitole dokládá, že kapacita krycích objektů v kontextu implementovaného stegosystému řádově odpovídá *Hydan*u.

Zadání práce lze považovat za naplněné.

Literatura

[1] Executable and Linkable Format (ELF). Tool Interface Standards (TIS), [Online; navštíveno 14.11.2017].
URL http://www.skyfree.org/linux/references/ELF_Format.pdf

[2] Microsoft Windows Bitmap File Format Summary. [Online; navštíveno 5.5.2018].
URL https://www.fileformat.info/format/bmp/egff.htm

[3] Definition of steganography in English by Oxford Dictionaries. 2018, [Online; navštíveno 5.5.2018].
URL https://en.oxforddictionaries.com/definition/steganography/

[4] Anckaert, B.; Sutter, B. D.; Chanet, D.; aj.: Steganography for Executables and Code Transformation Signatures. In *Information Security and Cryptology – ICISC 2004*, Springer Berlin Heidelberg, 2005, ISBN 978-3-540-32083-8, s. 425–439.

[5] Betancourt, S. R.: Steganography: A New Age of Terrorism. 2004, [Online; navštíveno 5.5.2018].
URL
https://www.giac.org/paper/gsec/3494/steganography-age-terrorism/102620

[6] Boxer, A.: Trithemius Redivivus. 2015, [Online; navštíveno 5.5.2018].
URL http://trithemius.com/

[7] Committee, D. D. I. F.: DWARF Debugging Information Format Version 4. 2010, [Online; navštíveno 12.5.2018].
URL http://www.dwarfstd.org/doc/DWARF4.pdf

[8] Easttom, C.: *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. McGraw-Hill Education, 2015, ISBN 9781259588099.
URL https://books.google.cz/books?id=FxKhCgAAQBAJ

[9] El-Khalil, R.: Hydan, information hiding in program binaries. 2003, [Online; navštíveno 30.1.2018].
URL http://www.crazyboy.com/hydan

[10] Fkirin, A.; Attiya, G.; El-Sayed, A.: Steganography Literature Survey, Classification and Comparative Study. *Communications on Applied Electronics (CAE)*, ročník 5, ISSN 2394-4714.

[11] Foundation, P. S.: The Python Profilers – Python 3.6 documentation. [Online; navštíveno 15.5.2018].
URL http://docs.python.org/3.6/library/profile.html

[12] Foundation, P. S.: re – Regular expression operations – Python 3.6 documentation. [Online; navštíveno 13.5.2018].
URL http://docs.python.org/3.6/library/re.html

[13] Žilka, R.: *Steganografie a stegoanalýza*. Diplomová práce, Masarykova univerzita, Fakulta informatiky, 2008, vedoucí práce: Ing. Mgr. Zdeněk Říha, Ph.D.

[14] International, E.: ECMA-404 2nd Edition: The JSON Data Interchange Syntax. 2017, [Online; navštíveno 13.5.2018].
URL http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[15] Johri, P.; Das, S.; Mishra, A.; aj.: A review on steganography and cryptography. In *Computing for Sustainable Global Development (INDIACom)*, IEEE, 2016, ISBN 978-9-3805-4421-2.

[16] Joyce, J.: Steganography? *Scientific Computing & Instrumentation*, 2002, ISSN 1524-2560.

[17] Kipper, G.: *Investigator's Guide to Steganography*. CRC Press, 2003, ISBN 9780203504765.
URL https://books.google.cz/books?id=qGcum1ZWkiYC

[18] Lejska, K.: X86 Opcode and Instruction Reference. [Online; navštíveno 14.11.2017].
URL http://ref.x86asm.net

[19] Mixeur: X86 cpus' Guide. [Online; navštíveno 12.5.2018].
URL http://www.x86-guide.com

[20] Mrdovic, S.; Perunicic, B.: Kerckhoffs' principle for intrusion detection. In *Networks 2008 - The 13th International Telecommunications Network Strategy and Planning Symposium*, 2008, doi:10.1109/NETWKS.2008.6231360.

[21] Noé, A.: Steganography: A New Age of Terrorism. 2009, [Online; navštíveno 5.5.2018].
URL https://www.matroska.org/files/matroska.pdf

[22] OSDev.org: X86-64 Instruction Encoding – OSDev Wiki. [Online; navštíveno 15.5.2018].
URL http://wiki.osdev.org/X86-64_Instruction_Encoding

[23] Sharma, S.; Gupta, A.; Trivedi, M. C.; aj.: Analysis of Different Text Steganography Techniques: A Survey. In *2016 Second International Conference on Computational Intelligence Communication Technology (CICT)*, 2016, ISBN 978-981-10-6426-5, ISSN 1865-0929, s. 130–133.

[24] Singh, N.: Survey Paper on Steganography. *International Refereed Journal of Engineering and Science (IRJES)*, ročník 6, 2017, ISSN 2319-1821.